컴퓨터 소프트웨어 캡스톤 PBL(인공지능 기반 악성코드 분석)

악성 파일 분석 & 기계학습을 통한 악성코드 분류

2018008059 김은수

2018008686 오하은

목차

1. 프로젝트 개요

1.1 프로젝트 명

악성 파일 분석 및 기계 학습을 통한 악성코드 분류이다.

1.2 프로젝트 목적

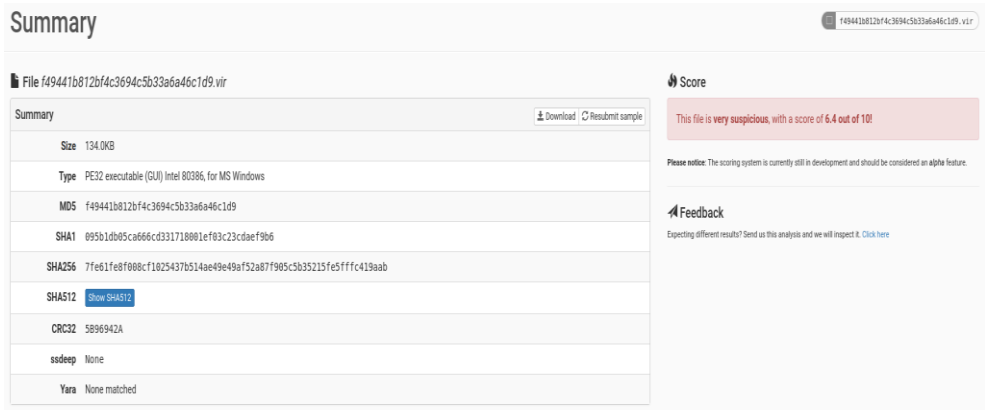
- Cuckoo sandbox를 통해서 악성파일과 정상 파일을 분석한다.
- 기계학습을 기반으로 악성파일과 정상파일을 분류하는 모델을 생성한다.

2. 악성 파일 분석

악성파일 및 정상파일을 Cuckoo sandbox에 넣고 Cuckoo sandbox를 통해서 분석을 수행한다.

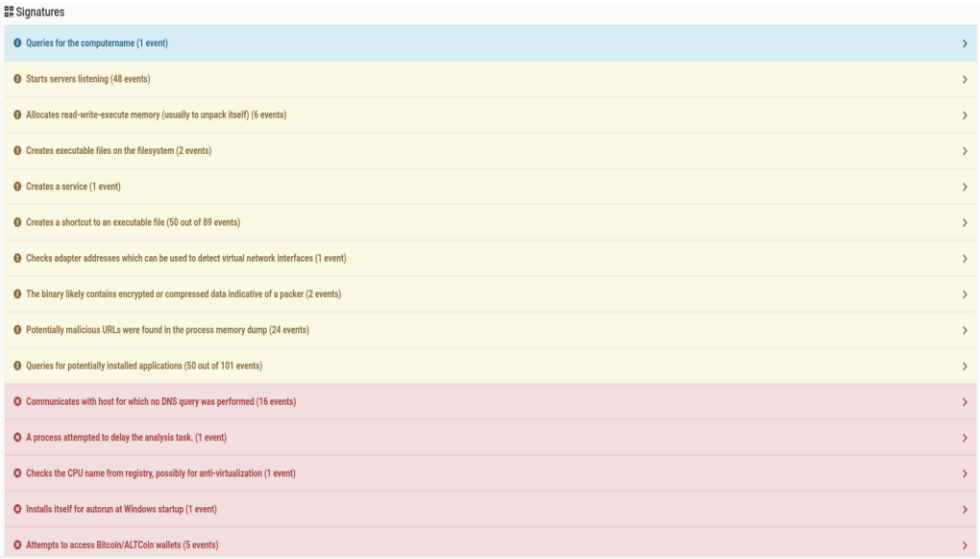
- 파일을 넣으면 해당 파일의 위험도를 score로 보여준다. Ground truth label과 비교해봤을 때 score가 2이상이면 악성파일이다.

- Summery를 통해 파일의 크기, 파일 타입, MD5, SHA1, SHA256등 기본적인 파일의 정보를 볼 수 있다. 해시 값은 이전에 이와 같은 악성코드는 없는 지 검색해볼 수 있다.

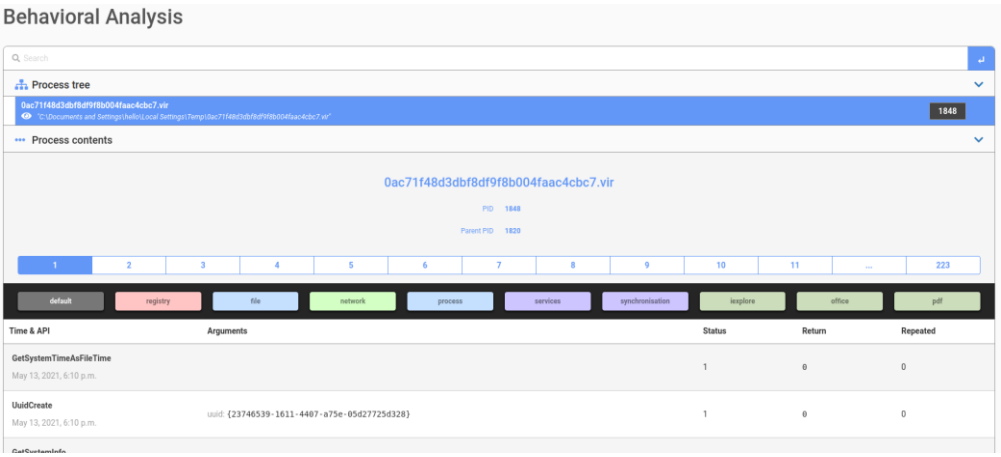


- Signature는 악성코드의 종류나 행위를 간략하게 보여주는 부분이다.

  Ex. 0ac71f48d3dbf8df9f8b0004faac4faac4cbc7.vir의 signature



- Behavioral analysis는 악성코드에 대한 동적 분석을 보여주는 부분으로 악성코드를 실행시켜 돌아가는 과정을 보여주기 때문에 악성코드가 하는 행동을 찾아낼 수 있다. 0ac71f48d3dbf8df9f8b0004faac4faac4cbc7.vir 파일의 동적 분석 결과이고 registry에 해당되는 내용을 살펴보겠다.



- Registry를 보면 RegCreateKey, RegSetValue를 이용하여 Run 레지스트리에 0ac71f48d3dbf8df9f8b0004faac4faac4cbc7.vir 파일을 등록시켜 윈도우 실행 시 자동 실행되도록 하고 있다. 이는 signature에서 나온 "install itself for autorun at window startup"에 해당되는 내용이다.

RegCloseKey May 13, 2021, 6:10 p.m.	key_handle: 0x000000f0	1	0	0
RegCreateKeyExA May 13, 2021, 6:10 p.m.	regkey_r: SOFTWARE\Microsoft\Windows\CurrentVersion\Run base_handle: 0x80000002 key_handle: 0x000000f0 class: options: 0 access: 0x00020006 disposition: 2 regkey: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	1	0	0
RegSetValueExA May 13, 2021, 6:10 p.m.	key_handle: 0x000000f0 regkey_r: AmdAgent reg_type: 1 (REG_SZ) value: C:\Documents and Settings\hello\Local Settings\Temp\0ac71f48d3dbf8df9f8b004faac4cbc7.vir regkey: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\AmdAgent	1	0	0

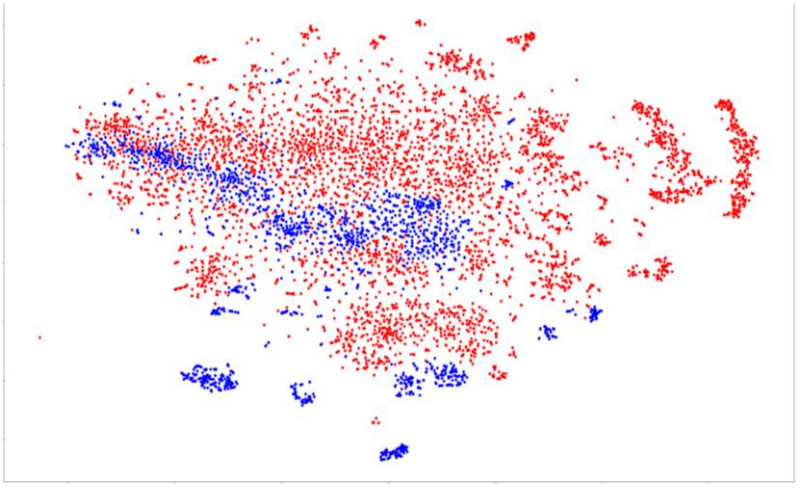
3. 기계학습을 통한 악성/정상파일 분류

3.1 서론

악성 파일 내 API 호출 순서가 악성코드를 판단하는 데 중요한 feature가 된다는 연구(참고 1)를 참고하여 악성파일 API 호출 순서를 통해서 악성파일과 정상파일을 분류하는 모델을 생성해보고자 하였다. Cuckoo sandbox를 통해 API을 추출하였고 4개의 머신 러닝 기법을 사용하여 학습을 하여 정확도 약 87%의 모델을 만들었다.

3.2 악성코드 특성 추출

- Dec2Vec을 활용하여 악성파일, 정상파일 간 차이 시각화



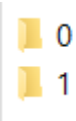
API 호출 순서를 하나의 문서로 생각하여 Doc2Vec을 활용하여 시각화 해보았다. 파란색 도트(정상 파일), 빨강색 도트(악성코드)가 소, 대규모로 군집을 이루고 있음을 확인할 수 있고 이를 통해 동적 API 시퀀스를 악성파일과 정상파일을 구분하는 feature로 사용가능 할 것이라 판단하였다.

- Report.json 파일에서 API 추출

```
for first in json_data['signatures']:
    marks = first.get('marks')
    for list in marks:
        call = list.get("call")
        if call != None:
            print(call.get("api"))

first = json_data["behavior"]
processes = first["processes"]
for li in processes:
    calls = li["calls"]
    if calls:
        for di in calls:
            print(di["api"])
```

```
WSAStartup
GetCursorPos
CoInitializeEx
GetTempPathW
GetTempPathW
LdrGetDllHandle
LdrGetDllHandle
GetSystemInfo
NtQuerySystemInformation
NtAllocateVirtualMemory
NtAllocateVirtualMemory
LdrLoadDll
```

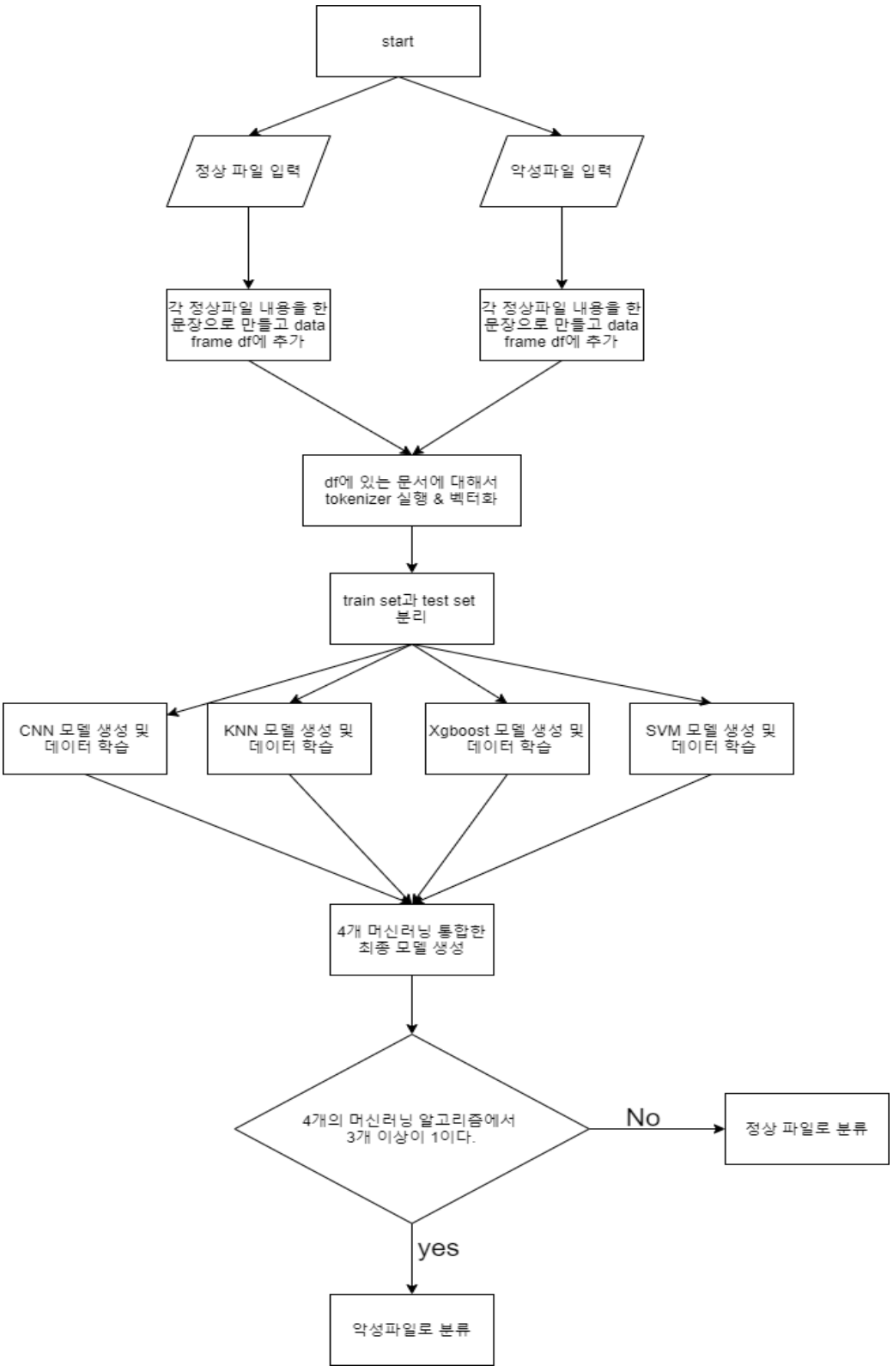
cuckoo sandbox에서 분석되는 파일마다 report.json이라는 파일이 존재한다. 이 파일 내부에는 파일의 정보들이 json 형태로 저장되어 있고 이 json 파일에서 API call 부분을 추출하여 API 호출 순서대로 하나의 문서로 보고 txt 파일 형태로 저장하였다.



그 다음에 악성파일은 1폴더에 정상파일은 0폴더에 저장하였다.

3.3 탐지 알고리즘 구성

3.3.1 순서도



3.3.2 환경

- 우분투 16.04
- Cuckoo sandbox 2.0.4
- Tensorflow 2.5.0
- Python 3.7.10

3.3.3 KNN 알고리즘

- 알고리즘 설명:

K-NN알고리즘은 데이터로부터 거리가 가까운 k개의 다른 데이터의 레이블을 참조하여 분류하는 알고리즘으로 거리를 측정할 때 유클리디안 거리 계산법을 사용한다.

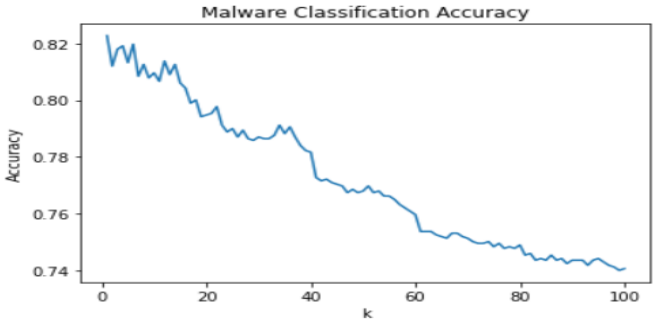
- 코드 설명

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import KNeighborsClassifier

knn_model = KNeighborsClassifier(n_neighbors=1)
knn_model.fit(X_train, y_train)
accuracy = knn_model.score(X_test,y_test)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Sklearn 모듈을 활용한다. KNN classifier 모델을 사용한다. KNN은 test 문서가 들어왔을 때 이 test 문서와 가장 가까이에 있는 k개의 training data 중 과반수의 label로 test 문서를 labeling해준다.

K를 가장 적절한 값으로 설정하기 위해 k를 1부터 100까지 for문을 돌려서 정확도를 구해보고 이를 시각화 해보았다.

```
k_list = range(1,101)
accuracies = []
for k in k_list:
    classifier = KNeighborsClassifier(n_neighbors=k)
    classifier.fit(X_train, y_train)
    accuracies.append(classifier.score(X_test,y_test))

plt.plot(k_list, accuracies)
plt.xlabel("k")
plt.ylabel("Accuracy")
plt.title("Malware Classification Accuracy")
plt.show()
```



K가 10 이내여야 성능이 좋을 것으로 판단되었다. 따라서 Knn_model = KNeighborsClassifier(n_neighbors=1)를 통해 모델에서는 K를 1로 설정하였다.

- 결과

K가 1일 때 83.85%의 정확도가 나왔다.

```
Accuracy: 83.85%
```

K가 3일 때 81.7%의 정확도가 나왔다.

```
Accuracy: 0.817965496728138
```

K가 5이내일 때 80% 내외의 정확도가 나오는 걸 확인할 수 있었다.

3.3.4 CNN

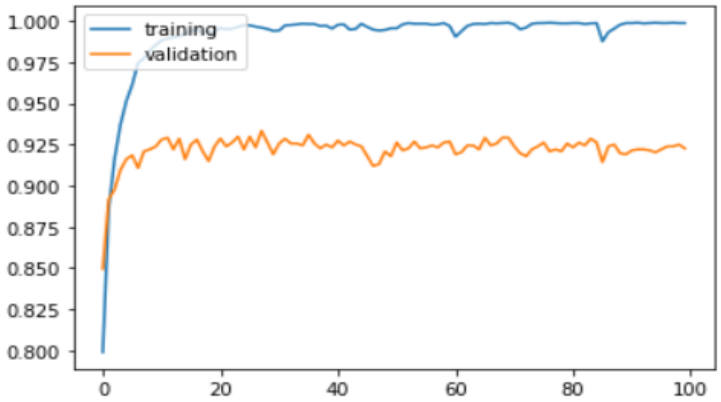
- 알고리즘 설명: CNN은 이미지에 주로 쓰이는 알고리즘으로 이미지에 대해서 픽셀별로 합성 곱을 통해 feature를 뽑아내고 이를 가지고 classification에 사용한다. 이런 CNN을 이 모델에 사용한 이유는 CNN 이 근처 벡터들을 묶어서 합성곱을 하기 때문에 우리가 사용하는 시퀀스가 있는 API에도 적용시킬 수 있을 것이라 생각했다. N-gram과 유사하게 묶여서 들어가기에 근처에 나타나는 API를 알 수 있고 이는 순서의 의미를 담았다고 할 수 있다.

- 코드 설명

```
from keras.models import Sequential
from keras import layers
from keras.layers import Dense, Dropout, Activation, Conv1D, MaxPooling1D, Embedding, Flatten
from keras import optimizers
embedding_dim = 100

def cnn1():
    model = Sequential()
    model.add(layers.Embedding(vocab_size, embedding_dim, input_length=maxlen))
    model.add(layers.Conv1D(128, 5, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(10, activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model
cnn_model = cnn1()
history = cnn_model.fit(X_train, y_train,
                    epochs=100)
```

Keras 모듈을 사용하였다. 구성한 CNN은 Convolutional layer는 하나이고 max pooling layer가 존재한다. Optimizer로 Adam optimizer를 사용하며 loss 함수는 binary cross entropy를 사용한다. 학습은 epoch을 100으로 설정하여 전체 데이터를 100번 학습하였다.

```
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['training', 'validation'], loc = 'upper left')
plt.show()
```



Epoch에 따른 정확도를 살펴본 결과 validation set이 0.9이상의 정확도를 가지다는 것을 확인할 수 있었다.

- 결과

```
Accuracy: 93.03%
```

정확도는 93%가 나왔다.

3.3.5 Xgboost 알고리즘

- 알고리즘 설명: Xgboost는 여러 개의 Decision Tree를 조합해서 사용하는 Ensemble 알고리즘으로 Gradient Boosting 알고리즘을 분산환경에서도 실행할 수 있도록 구현해 놓은 라이브러리다. Regression, Classification 문제를 모두 지원하며, 성능과 자원 효율이 좋아서, 인기 있게 사용되는 알고리즘이다.
- 코드 설명:

  Bayesian Optimization을 활용하여 Xgboost의 튜닝 값을 조절하였다.

  Bayesian optimization을 위한 파라매터는 다음과 같다.

```
ranges = {
    'numRounds': (1000, 2000),
    'eta': (0.03, 0.1),
    'gamma': (0, 10),
    'maxDepth': (4, 10),
    'minChildWeight': (0, 10),
    'subsample': (0, 1),
    'colSample': (0, 1),
    }
```

```
1 max_params = bayesOpt(X_train, y_train) #max_params = 최적의 파라미터
```

```
|   iter    |  target   | colSample |    eta    |   gamma   | maxDepth  | minChi... | nu
-------------------------------------------------------------------------------------------
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 1         | -0.2995   |  0.5876   |  0.07342  |  2.806    |  5.803    |  7.515    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 2         | -0.3646   |  0.3708   |  0.0407   |  5.425    |  7.077    |  8.994    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 3         | -0.3295   |  0.7423   |  0.08254  |  5.075    |  6.629    |  8.046    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 4         | -0.303    |  0.7533   |  0.06241  |  2.378    |  9.913    |  3.046    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 5         | -0.3728   |  0.08999  |  0.08328  |  9.064    |  4.621    |  9.905    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 6         | -0.3537   |  0.3222   |  0.09389  |  5.353    |  9.157    |  0.1163   |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 7         | -0.2468   |  0.1231   |  0.06362  |  0.8239   |  5.035    |  2.518    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
| 8         | -0.3845   |  0.5046   |  0.07918  |  3.762    |  9.675    |  1.685    |  1
params {'objective': 'reg:linear', 'booster': 'gbtree', 'eval_metric': 'mae', 'tree_meth
```

bayesOpt 함수로 최적화된 파라미터를 찾고 찬긴 최적화된 파라미터를 이용하여 xgboost를 학습시켰다.

```
params=(
    n_estimators=1980,
    eta=0.04268380464143151,
    max_depth=9,
    subsample=0.27562492851302745,
    objective= 'reg:linear',
    silent= 1,
    min_child_weight=7,
    gamma=0.3429412218176964,
    booster='gbtree',
    eval_metric='mae',
    tree_method='auto',
    scale_pos_weight=0.48
)
```

```
1 dtrain = xgb.DMatrix(X_train, label=y_train)
2 booster = xgb.train(max_params, dtrain, num_boost_round=1230)
3
4 dtest = xgb.DMatrix(X_test)
5 y_pred = booster.predict(dtest)
6 y_pred = y_pred > 0.5
7 y_pred = y_pred.astype(int)
8 accuracy = accuracy_score(y_pred, y_test)
9 print("Accuracy: %.2f%%" % (accuracy * 100.0)) # 예측률: 91.58%
```

- 결과

약 91.58%의 정확도가 나왔다.

3.3.6 SVM 알고리즘

- 알고리즘 설명

SVM은 Support Vector Machine의 약어로 Support Vector와 hyper-plane이 주요 개념인 머신 러닝 알고리즘이다. 신경망에 비해 간결하면서 뛰어난 성

능을 보여서 많이 사용된다. SVM은 분류나 회귀 분석에 사용이 가능하며, 특히 분류 쪽의 성능이 뛰어나다. SVM은 마진이 크면 클수록 학습에 사용하지 않은 새로운 데이터가 들어오더라도 잘 분류할 가능성이 커지기 때문에 최고의 마진을 가져가는 방향으로 분류를 수행한다.

- 코드 설명

```
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

svm_model = SVC(kernel='linear', C=1000)
svm_model.fit(X_train, y_train)
svm_y_pred = svm_model.predict(X_test)

accuracy = accuracy_score(y_test, svm_y_pred)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

kernel을 사용해서 데이터를 높은 차원으로 이동시켜 분리했으며 커널 값은 선형을 사용했다. 마진 값이 작더라도 데이터셋을 정확히 분류하길 원해 C값을 1000으로 설정하였다.

- 결과

  약 87%의 정확도가 나왔다.

4. 모델 완성

해당 파일을 넣었을 때 각 알고리즘을 통해서 나온 값을 도출하고, 2개 이상의 모델에서 1이 나온다면 (악성파일이 나온다면) 악성파일로 판단하는 모델을 만들었다. 지난 최종 발표 시간에는 전체 test data를 사용해서 모델을 돌렸을 때의 성능을 구하지 않았지만, 보고서를 작성하면서 4개의 알고리즘을 assemble했을 때의 test data의 정확도를 구해보았다.

- 코드 설명

```
for i in range(len(X_test)):
    print(i)
    testing = []
    # CNN
    cnn_pred_test = np.float(cnn_pred[i])
    if cnn_pred_test > 0.5:
        print("CNN : 1")
        testing.append('1')
    else:
        print("CNN : 0")
        testing.append('0')

    # KNN
    knn_pred_test = knn_pred[i]
    if knn_pred_test > 0.5:
        print("KNN : 1")
        testing.append('1')
    else:
        print("KNN : 0")
        testing.append('0')

    # SVM
    svm_pred_test = svm_pred[i]
    if svm_pred_test > 0.5:
        print("SVM : 1")
        testing.append('1')
    else:
        print("SVM : 0")
        testing.append('0')
```

```
# Xgboost
xgb_pred_test = xgb_pred[0]
if xgb_pred_test > 0.5:
    print("Xgboost : 1")
    testing.append('1')
else:
    print("Xgboost : 0")
    testing.append('0')

if testing.count('1') > testing.count('0'):
    print("This file is Malware")
    answer.append(1)
else:
    print("This file is Normal")
    answer.append(0)
```

각 알고리즘을 활용해서 예측했을 때 예측 값이 0.5보다 크면 악성파일로 간주한 후 2개의 알고리즘에서 해당 파일이 악성파일이라고 예측했다면 전체적으로 악성파일이라고 판단하였다. 이처럼 2개의 알고리즘으로 판단하였을 때는 성능이 80% 정도 나왔다.

그러나 모델의 성능을 구하는 과정에서 4개의 알고리즘 중 3개 이상의 모델이 1이 나오면 악성파일로 판단했을 때 성능이 약 86.77%로 더 높게 나왔다. 따라서 3개이상의 머신 러닝 알고리즘에서 해당 파일이 악성파일이라고 나와야 악성파일이라고 분류해주는 모델로 제작하였다.

5. 논의 및 결론

- 참고한 논문에서는 2-gram을 추출하여 특징벡터로 2-gram+API를 사용하였다. 다양한 방식으로 시퀀스를 벡터화하여 머신 러닝을 진행하면 더 성능이 높아질 수 있을 것이다.
- 동적분석으로 얻은 API외에 정적분석에서 얻은 feature들을 같이 사용하면 더 좋은 성능이 나올 수 있을 것이다.
- SVM, Xgboost 알고리즘이 시간이 오래 걸린다는 단점이 있다.
- 단순한 머신 러닝 알고리즘을 사용했지만 성능이 높게 나옴을 확인할 수 있었고 머신 러닝이 악성파일을 분류하는 데 유용하게 사용할 수 있을 것이라 생각된다.

6. 참고논문

- 이현종, 허재혁, 황두성, "악성코드 탐지를 위한 기계학습 알고리즘의 성능 비교", (단국대학교 소프트웨어학과, 컴퓨터공학과, 2018), 한국컴퓨터정보학회 동계학술대회 논문집 제26권 제1호 (2018. 1)